Tactical Multi-Unit Pathplanning with GCLS

Alexander Nareyek and Aditya Kristanto Goenawan
Department of Electrical & Computer Engineering. National University of Singapore, Singapore 117576
alex@ai-center.com, aditya.kristanto@gmail.com

Abstract

In this paper, we are considering advanced pathplanning problems that feature finding paths for multiple units subject to rich path constraints. Examples of richer constraints are the following of other units or to stay out of sight of a specific unit. Little attention has so far been given to richer pathplanning problem where the objective is more than reaching a specific destination from a starting point such that the path length is minimized. Richer pathplanning problems occur in many complex real-world scenarios, ranging from computer games to military movement planning. In this paper, a novel way to formally specify such problems and a new local-search strategy to solve such problems are proposed and demonstrated by a prototype implementation. Among the design goals are real-time computability as well as extendibility for new constraints and search heuristics.

Introduction

Pathplanning problems are encountered in various fields, such as robotics, military simulations and electronic gaming. Occasionally, the pathplanning problems encountered in those fields are not as simple as trying to find the shortest path to a specific location, but involve richer pathplanning constraints. For example, in the electronic gaming field, a computer-guided character/unit A might need to avoid another computer-guided unit B while maneuvering to area X. Or, a group of three units A, B and C might need to trap another unit D (not allowing the unit to move without colliding) within a specific time window. The paths of the units are often dependent on the other units' paths here.

Common pathplanning algorithms, such as A* (Hart, Nilsson, and Raphael 1968) and its variants, such as D* (Silver 2006), or Particle Swarm Optimization (Mohamed et al. 2010), and Ant Colony Optimization (Eghbali and Sharbafi 2010), cannot be used to tackle such richer pathplanning problems. These algorithms are specialized for a single-unit pathplanning that optimize a path for a specific start and destination location. The richer pathplanning problems discussed in this paper involve multiple units and complex constraints restricting potential paths, might not even specify a specific start and destination location for a unit, and don't necessarily have a path minimization goal. In fact, in the general case, TMPPs are undecidable because of the arbitrary length of a path for a unit and costs that might be independent of the path length.

Such more complex pathplanning problems, subsequently called *Tactical Multi-Unit Pathplanning (TMPP)* problems, were first tackled by (Wang, Malik, and Nareyek 2009), who utilized a backtracking-based approach mixed with A*. A* requires continuously increasing costs toward the goal, and only a subset of scenarios was thus tackled. In addition, their specifications and methods are neither easily extendable nor particularly real-time focused. In this paper, a special focus is on an extendable specification format and real-time/anytime methods for solving TMPP problems, based on the concept of using global constraints for local search (GCLS) (Nareyek 2001).

Problem Specification Format

The specification format described in this section tries to strike the balance between extendibility and complexity. The specification format does not incorporate domain-specific knowledge, thus making it generally usable across many domains and algorithms.

On a high conceptual level, a TMPP problem is a general search problem, and its specification can be divided into:

- Options: Options refer to a set of specifications that define/unfold the search space, e.g., the units involved, location vertices and connecting edges, and the abilities that a unit can perform.
- Constraints: Constraints refer to a set of specifications that cut the search space size down and need to be fulfilled. For example, to specify a unit's movement start and destination, along with collision avoidance requirements, location constraints and a collision constraint can be defined. Apart

from specifications that need to be fulfilled, a constraint can also express preferences among feasible solutions (for optimization), like a minimal path length for a unit.

Loosely following paradigms like constraint programming, this specification format is very variable and allows to easily vary problem specifications. For new application domains, the expressiveness can also easily be extended by adding new constraint types etc. to the specification language.

To make the definition of options and constraints above more compact, we also use tags to refer to a set of specification objects. For example, constraints can be applied in bulk by grouping the units and vertices together by using group and area tags respectively.

In the following, we will discuss some basics for options, constraints, and solutions for TMPP problems. A formal specification format is not given, as the requirements for such a language might depend very much on the target platform and application (though a concrete specification format would be a relatively straight-forward task). This paper focuses on the principles and is not meant to suggest a particular implementation.

Options

Options are further divided into world, edge and unit options.

World Options

Pathplanning can be done for very different map representations, e.g., the world can be a grid world or a continuous world, a 2D world or a 3D world. Depending on the world configuration, more parameters, such as width/length grid dimensions can be specified.

World options also include the specification of a time representation, e.g., discrete time versus continuous time, and potential additional global parameters like a start time and end time for the paths.

Edge Options

The common representation of movement possibilities is a graph with location vertices and either uni- or bi-directional connection edges. The usage of a graph representation allows the world type to be abstracted from the world options, i.e., a 2D-grid and 3D-continuous world both use the same vertex representation. We vary this approach slightly, where single locations are represented as a set of vertices and directed edges. We require that every unit is traveling on an edge at all times, and this can for example be used for waiting or other non-movement-related activities, and to unify collision checks by being concerned with edges only.

Edge options define the existence of vertices and edges. We only call them 'edge options' to emphasize the dominant role of edges. Additional properties can be set for vertices and edges, indicating accessibility for specific units, times of accessibility, terrain types, or other information that affects path generation.

Unit Options

Unit options define the existence of units and their properties. Properties are features that are used by constraints to create and check paths, e.g., defining that a unit can move, stop, see, taunt and attack, or that a unit has a minimum speed of 0.0 and maximum speed of 5.0.

As a side note, properties do not need to specify single values but can be more complex objects, e.g., also adding edge and time information to a vision property to specify that a villager can only see during daytime and near light sources during nighttime.

Tags

Vertices, edges and units can also be tagged to belong to location or unit groups, such as City or Street, or Farmers or Guards. Groups can be used in definitions in the same way as single vertices, edges and units. This tagging is useful to compress the specifications because many properties and constraints often apply to similar object sets.

Constraints

A pathplanning application will support a set of constraint types, such as a Collision constraint or a Following constraint. A problem specification includes instances of these types, initialized with constraint-specific parameters like units, vertices and edges. For example, a Following constraint instance can be created to affect unit A and unit B for a time range of (0,25).

A constraint type may only support particular world options, and not be able to read/understand all property specifications of units, vertices and edges. While the support of all specified world options is required, a constraint may be used even if it's not able to process all property specifications. This means that only the existence of units, vertices, edges, the time representation, and an edge assignment for all units spanning the solution duration and observing possible edge connections, serve as hard-coded/structural limitations to possible solution exploration. All other constraints (like observing speed limits) need to be explicitly added as constraint instances.

A constraint instance may post – potentially multiple – satisfaction and optimization values. Satisfaction values express how much the core constraint is away from being fulfilled, i.e., how far the current unit paths are from being executable. For example, a Collision constraint may post the cumulative collision durations for the current paths of all units as costs. Optimization values express the quality of the current unit paths, i.e., the paths might be executable from the constraint's point of view, but there may still be room for improvement. For example, the Collision constraint may post the minimal distance between all units as an optimization value to create buffer zones in case of dynamic changes of the situation.

We call all of the optimization and satisfaction values 'costs' in the following, where the goal is to bring satisfaction values to 0, and to maximize optimization values. There can be multiple satisfaction and optimization costs per constraint (for example in case of multi-objective optimization), and each available cost value of a constraint type needs to be associated with a unique cost type identifier (like COLLISIONC-MINDISTANCE).

As a side note, there may be constraints like PathLength, but in most application areas, this might actually not be very useful. Constraints like RealisticPath or ConserveEnergy might be much more appropriate.

Solutions

A *potential solution* includes, for each unit, a sequence of connected edges spanning the complete time horizon to be planned for. Edges in these sequences may be enhanced with properties, e.g., to indicate movement parameters or activities to be executed by a unit.

In addition to these edge sequences for the units, a potential solution also includes the cost information of all constraint instances. If all satisfaction costs are 0, a *potential solution* is also a *solution*.

Solving Strategy

TMPPs can – like other search problems – be tackled with a variety of search methodologies. Its complexity is in the worst case undecidable if time is unrestricted, or if time is continuous and there is no minimum of edge movement times and no maximum of property enhancements for edges. However, such a discussion is purely academic and of no practical relevance. In practice, we are interested in goals like the best possible solution given some time limit, or even the best possible potential solution in case a solution cannot be constructed.

The two dominant search approaches are refinement search and local search. In refinement search, a solution is stepwise constructed (e.g., movement by movement toward full paths), and backtracking utilized if search gets into a

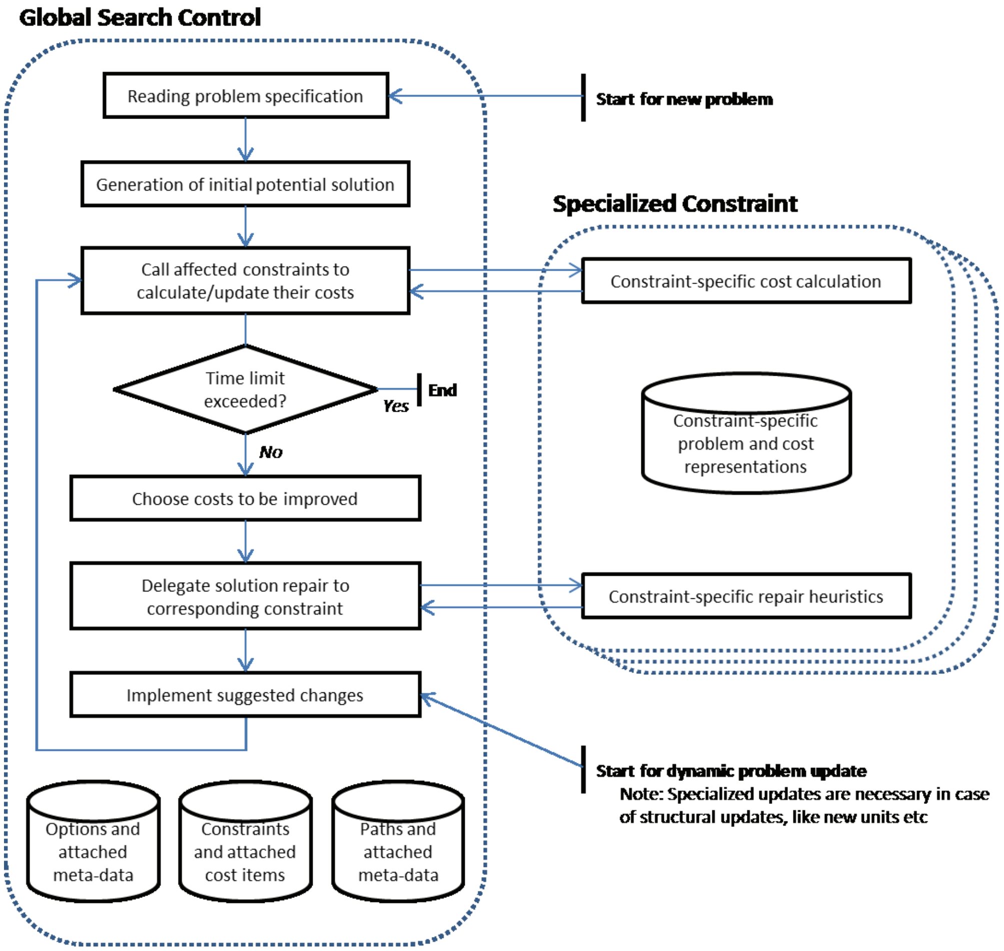
Global Search Control
Reading problem specification
Start for new problem
Generation of initial potential solution
Specialized Constraint
Call affected constraints to calculate/update their costs
Constraint-specific cost calculation
Time limit exceeded?
Yes
End
No
Constraint-specific problem and cost representations
Choose costs to be improved
Delegate solution repair to corresponding constraint
Constraint-specific repair heuristics
Implement suggested changes
Start for dynamic problem update
Note: Specialized updates are necessary in case of structural updates, like new units etc
Options and attached meta-data
Constraints and attached cost items
Paths and attached meta-data

Fig. 1. The overall search process.

dead-end. In local search, a fully grounded potential solution is iteratively changed toward the best solution (e.g., changing a unit's path a bit such that there are fewer collisions). A discussion of the pros and cons of both approaches is beyond this paper (e.g., see (Nareyek 2001) for a discussion), and it is sufficient to say that real-time properties and the ability to easily incorporate dynamic situation changes led us to the choice of local search, and more specifically GCLS (Nareyek 2001). Figure 1 shows the overall solving process.

Initial Potential Solution Generation

Local search works by making changes to an existing, fully grounded potential solution. Thus, to get the process started, we need to generate an initial potential solution.

It may be straight-forward to generate a random initial potential solution, but to speed up the search process, it is useful to already consider some of the constraints in this process. For example, if the units' start and end positions are specified by Location constraints, we can simply run A* to get the initial paths (which is used in the prototype system described later).

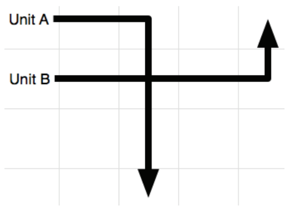

Fig. 2. The initial solution generation can for example be based on A by using the Location constraints for the units' start and end locations (if such constraints are existing).*

Constraints as Evaluators and Optimizers

In the GCLS approach, constraints serve as evaluators to compute costs as well as optimizers to make modifications to a potential solution in order to optimize it. This follows the principle that costs should be tackled by the objects that have the deepest insights about the specific costs and their causes. Domain-specific knowledge related to the constraints' cost reduction is explicitly encouraged to be integrated.

Constraints should not re-evaluate their costs in each iteration but only if a change potentially affects them. They should thus subscribe to specific component patterns of the current solution, and only be called for updates in case there is a match.

Constraints can also use additional persistent internal data structures to more quickly evaluate costs and potential repairs.

A main component of a constraint is the improvement/repair heuristics, which suggest a change of the current potential solution. For example, a Collision constraint can contain two different heuristics. The heuristics will try to repair the paths of the affected units such that collisions are avoided. One simple way of repair is to make one unit wait for the other unit to pass before the unit moves on. Alternatively, it can repair the solution by providing an alternative path that avoids the collision for a unit involved in the collision (see Fig. 3). These two heuristics are used for the Collision constraint in the prototype system described later.

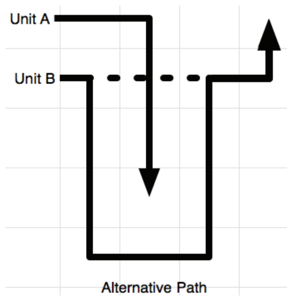

Fig. 3. The solution repair done by the Collision constraint. Initially unit B's path is the one indicated with dotted line (see Fig. 2). After repair, unit B avoids the collision via an alternative path.

To choose between the heuristics, the concept of non-stationary reinforcement learning (Nareyek 2003) can be incorporated. This concept chooses a heuristic with a random distribution based on the past performance. If heuristic H1 has been performing better than heuristic H2, then heuristic H1 is more likely to be chosen.

When a constraint is called for a solution repair, the interface should also include the possibility to specify resource restrictions for the computations. This is to make sure that only a specific amount of time/memory/etc is used for the computations, which is for example critical for the real-time responsiveness of the system.

Search-related Meta-Data

Purely relying on the independent repairs of constraints may not be a good idea because the constraints might be missing a lot of useful context information indicating which repairs might hurt or be beneficial for other constraints. We thus allow the constraints to attach meta-data to a specification's options and a potential solution's paths. Meta-data can follow the regular property definition format, and can be ignored by constraints in case they do not understand it.

For example, in the prototype system described later, constraints can attach cost information to edges that indicate how much additional cost or cost savings will be incurred if a specific unit uses the edge within a specific time interval. This meta-data will attract or repel constraints from using these edges for their repairs.

Choosing Costs to Be Improved

For the choice on the global level on which costs should get repaired, there are many alternatives as well. Depending on the goals, e.g., finding a possible solution as soon as possible versus finding the best solution in a given time versus finding pareto front solutions in multi-objective optimization, very different choice patterns might be applied. Furthermore, the process given in Fig. 1 only considers a sequential search process, while in the future, highly parallel solution explorations might be applied, e.g., by parallel repairs with interleaved integration phases and/or multiple repair evaluations and the choice of the best repair(s) for actual application. While still in a very preliminary state, a discussion for a similar approach can be found in (Kumar and Nareyek 2009).

From our experience in other domains, it works best to assign cost items based on cost types to specific cost collections, such as in the simplest case, one cost collection for satisfaction costs and one for optimization costs. Cost collections are processed in an absolute order, e.g., optimization costs are only considered once all satisfaction costs are repaired. Within a cost collection, cost items are selected relatively according their numeric values. Depending on the domain, various cost transformations might be useful for cost items before entering them into a cost collection, such as a scaling by specific factors. More sophisticated methods might take diversification/intensification patterns in search into account instead of choosing always the item with the highest costs (or a random choice in case of the optimization cost collection). These specifications should be passed as part of the option and constraint definitions in the problem specification.

A Prototype Implementation

As a proof-of-concept, we have implemented a system with a few basic constraint types (and very simple heuristics): a Location constraint type that requires a unit to be at a specific time interval on a specific edge, a Collision constraint type to prevent units utilizing edges at the same times, and a Meet constraint type that requires two units to be within a specific distance from each other over a specific time interval. The constraints have only satisfaction costs, i.e., only one cost collection is used, and a test run is terminated once costs of 0 are reached.

Some heuristics that were used for repairs were pointed out in the previous section, but we want to emphasize that these are surely not the best ones. The purpose of this paper is not to propose specific heuristics but to introduce the overall framework. The development of efficient constraints is a task for many years into the future, and specific application domains will surely require additional specialized constraints to fulfill advanced real-time requirements.

The system is implemented in Java, has about 5,000 lines of code, and the results below are obtained using a PC with an Intel i5 750 (2.67 GHz), 4GB DDR3 RAM, and the Java 6 SE Virtual Machine.

The results are always taken as the average of 1,000 runs between the 1,000th and 2,000th runs. After some tests, these runs are the runs whose runtime has become stable and negligibly affected by start-up-related processes.

Because of the limited space, we are providing only a very limited number of example cases for illustration.

Scaling

In this test, we are varying the number of constraints (same types) and the underlying maps and unit numbers.

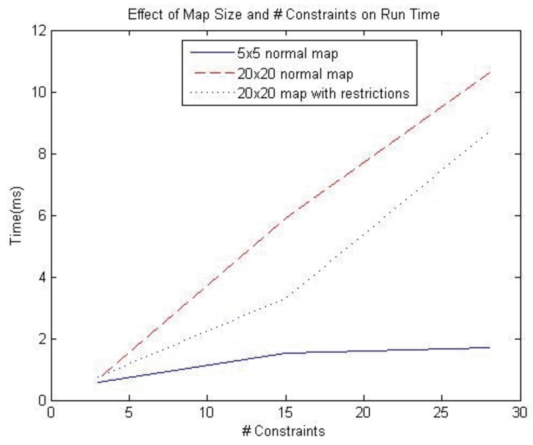

Fig. 5. The run time (until cost=0) scaling for different numbers of constraint instances on different maps.

"Normal map" refers to a fully accessible 2d-grid map with N/S/W/E edges, while "restricted map" refers to a map with many larger objects with inaccessible grid points.

The test scenarios are relatively easy, and Fig. 5 shows mainly linear growth, which is expected because of the increasing necessary cost evaluation updates. For smaller maps with short paths, this however hardly has an impact as the evaluations can be very quickly done.

Utilizing Meta-Data

In this section, the impact on utilizing meta-data (attaching cost information, as previously mentioned) is visualized. The test case used is a hard scenario with 7 units and 28 constraints. After 50 repair iterations, only very few runs reach costs of 0.

As visible in Fig. 6, meta-data often has a beneficial effect, especially for complex/hard scenarios, where not considering the interaction between constraints often otherwise has a negative effect on search performance.

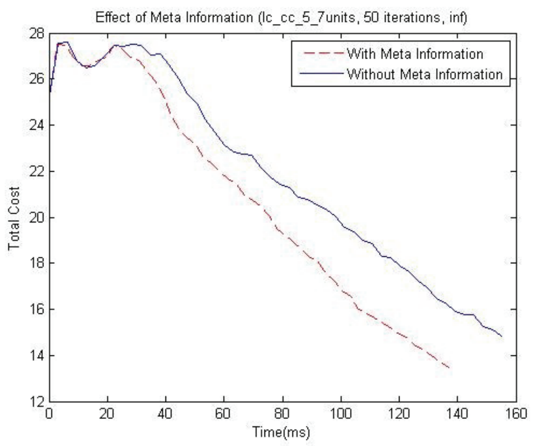

Fig. 6. The effect of utilizing meta-data.

Neighborhood Size

When looking for potential repairs, a constraint may have the option to either return a repair quickly, or to spend more time on exploring alternative repairs until the maximal iteration time is reached. It may appear as if a more intelligent exploration with an evaluation of a larger set of alternatives may generally be beneficial, especially for hard instances, but Fig. 7 indicates that this certainly does not hold in general (using a complex/hard scenario). Though we have to indicate that our test system's heuristics are relatively simple so far, which means that they might not be able to utilize the additional time effectively by generating enough really promising change alternatives.

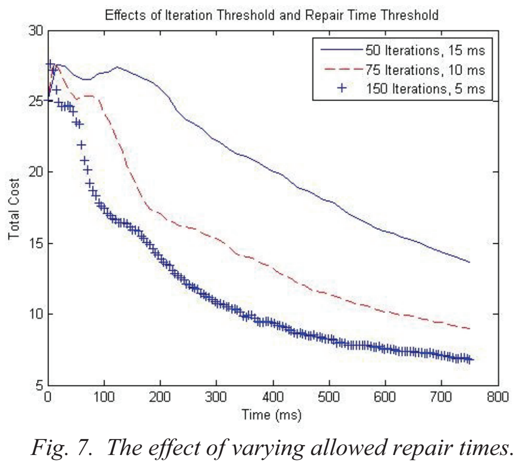

Fig. 7. The effect of varying allowed repair times.

Conclusion

We have presented a novel approach for the specification and solving of TMPPs that targets extensibility and real-time processing. New constraint types can easily be added to a system in a modular way, and the iterative search approach allows for interruptible/splittable processing while maintaining a current best solution. Dynamic changes of the problem specification (e.g., when the player in a computer game behaves differently than expected, or path execution of units has different results than expected) can also easily be integrated without the need for restarting search from scratch.

While the implementation of our proof-of-concept system leaves certainly a lot of room for improvement, we are convinced that the overall approach is the way to go forward, related to the general positive properties of the specification and search methods, and the results we got so far.

Because of the restricted space, we did not discuss much related work though; please refer to (Wang, Malik, and Nareyek 2009) for more discussions on TMPP-related work.

Acknowledgments

This work was supported by the Singapore National Research Foundation Interactive Digital Media R&D Program under research grant NRF NRF2007IDM-IDM002-051.

References

Eghbali, M., and Sharbafi, M.A. 2010. Multi agent routing to multi targets via ant colony. In Proceedings of the 2nd International Conference on Computer and Automation Engineering (ICCAE 2010), vol.1, 587-591.

Hart, P.E.; Nilsson, N.J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics 4(2), 100-107.

Mohamed, A.Z.; Sang Heon Lee; Aziz, M.; Hung Yao Hsu; and Ferdous, W.M. 2010. A proposal on development of intelligent PSO based path planning and image based obstacle avoidance for real multi agents robotics system application. In Proceedings of the 2nd International Conference on Electronic Computer Technology (ICECT 2010), 128-132.

Nareyek, A. 2001. Constraint-Based Agents - An Architecture for Constraint-Based Modeling and Local-Search-Based Reasoning for Planning and Scheduling in Open and Dynamic Worlds. Reading, Springer LNAI 2062.

Nareyek, A. 2001. Using Global Constraints for Local Search. In Freuder, E. C., and Wallace, R. J. (eds.), Constraint Programming and Large Scale Discrete Optimization, American Mathematical Society Publications, DIMACS Volume 57, 9-28.

Nareyek, A. 2003. Choosing Search Heuristics by Non-Stationary Reinforcement Learning. In Resende, M. G. C., and de Sousa, J. P. (eds.), Metaheuristics: Computer Decision-Making, Kluwer Academic Publishers, 523-544.

Kumar, A., and Nareyek, A. 2009. Scalable Local Search on Multicore Computers. In Proceedings of the Eighth Metaheuristics International Conference (MIC 2009), 146.1-146.10.

Silver, D. 2006. Cooperative pathfinding. In Rabin, S. (ed.), AI Game Programming Wisdom 3, Charles River Media, 99-111.

Wang, H.; Malik, O. N.; and Nareyek, A. 2009. Multi-Unit Tactical Pathplanning. In Proceedings of the 2009 IEEE Symposium on Computational Intelligence and Games (CIG 2009), 349-354.